



# Performance feature identification by comparative trace analysis

Daniel P. Spooner<sup>a,\*</sup>, Darren J. Kerbyson<sup>b</sup>

<sup>a</sup> *High Performance Systems Group, Department of Computer Science, University of Warwick, CV4 7AL, UK*

<sup>b</sup> *Performance and Architecture Laboratory (PAL), CCS-3, Los Alamos National Laboratory, NM 87545, USA*

Available online 7 January 2005

## Abstract

This work introduces a method for instrumenting applications, producing execution traces, and visualizing multiple trace instances to identify performance features. The approach provides information on the execution behavior of each process within a parallel application and allows differences across processes to be readily identified. Traces events are directly related to the source code and call-chain that produced them. This allows the identification of the causes of events to be easily obtained. The approach is particularly suited to aid in the understanding of the achieved performance from an application centric viewpoint. In particular, it can be used to assist in the formation of analytical performance models which can be a time-consuming task for large complex applications. The approach is one of human-effort reduction: focus the interest of the performance specialist on performance critical code regions rather than automating the performance model formulation process completely. A supporting implementation analyses trace files from different runs of an application to determine the relative performance characteristics for critical regions of code and communication functions.

© 2004 Elsevier B.V. All rights reserved.

**Keywords:** Performance modeling; High performance systems; Dynamic trace analysis; Performance visualization

## 1. Introduction

It is generally accepted that the peak-performance of high performance systems is the result of a complex interplay between the hardware architecture, the communication system and the applied workload. Knowledge of the processor design, memory hierarchy, inter-processor and network system, and workload arrange-

ment is necessary to understand the achievable performance.

As high-end computing facilities increase in scale and complexity, it is essential to consider the system performance throughout an architecture's life cycle: starting at the design stage where no system is available for measurement, through comparison of systems and procurement, to implementation, installation and verification, and finally to examine the effects of system updates over time.

A key approach that can be used at each stage of the life-cycle is to utilize detailed models that provide

\* Corresponding author. Tel.: +44 24 7657 3779;  
fax: +44 24 7657 3024.

E-mail address: [dps@dcs.warwick.ac.uk](mailto:dps@dcs.warwick.ac.uk) (D.P. Spooner).

information on the expected performance of a workload, given a particular architectural configuration. Depending on the complexity of the model, an expectation of the achievable performance of the workload can be obtained with reasonable fidelity.

The accuracy of a model, and hence, its effectiveness lie in its ability to capture an application's performance characteristics. It is considered advantageous to parameterize a model in terms of system configuration and calculation behavior, as this allows for the exploration of the performance space without being specific to a particular 'performance point'. Evaluation of anticipated scenarios such as the scaling effect of increasing the processor count, altering the workload size, or the impact of modifying the communication strategy can then be explored.

Performance models are widely used: from large-scale, tightly coupled systems through to dynamic and distributed Grid-based systems. For instance, performance modeling was used to: validate performance during the installation of ASCI Q at Los Alamos National Laboratory (LANL) [1] which ultimately lead to the system optimization resulting in a factor of two performance improvement [2]; compare the performance of large-scale systems such as the Earth Simulator [3]; and has been used in the procurement of many systems including ASCI purple (expected to be a 100Tflop machine). Performance models are also extensively applied in 'Grid' environments to consider service-orientated metrics in the provision of resource management services [4], and in the mapping of financial applications to available resources [5].

It is, however, generally acknowledged that the formation of a performance model is a complex task. It typically involves a thorough code analysis, inspection of important data structures, and analysis of profile and trace data. It can, therefore, be time-consuming to generate a detailed model given the large size of many scientific applications and the relative complexities of advanced data structures and highly optimized communication strategies. Historically, much of this work has been performed "by hand" using profilers and tracing libraries typically designed for fault analysis. As applications become more complex, the ability to facilitate the modeling process has become increasingly desirable.

Several semi-automated approaches have been proposed that aim to make the formation of a performance

model a simpler task using 'black-box' techniques in which individual performance aspects are observed but not necessarily understood. Examples include modeling the scaling behavior of basic-block performance [6] and modeling the memory behavior of basic-blocks and extrapolating to other systems [7]. These approaches tend to be specific to a particular processor configuration and/or application problem size.

In this work, we consider an approach that aims to simplify the process of generating a performance model, but not to automate it entirely. The purpose is not to simplify the resultant performance model, or detract from the skill-set required by the modeler, but rather remove unnecessary steps during formulation. While the answer to this question lies, in part, with the experience of the performance-modeler; we believe that tools can be developed (or adapted) to help locate and focus on the performance critical regions. Such regions are typically those whose execution behavior changes when the system configuration or application input-data is varied. Should a region of code not change (in the performance sense) to such configuration and input variations, it may be possible to approximate the behavior by a single timing for a given architecture. An essential part of the performance modeler's skill is distinguishing between the elements that require parameterization and those where a single-time will suffice.

While there are a number of post-analysis and diagnosis tools that can assist with identifying performance constraints such as identifying bottlenecks [8] or visualizing communication patterns [9], many are aimed at resolving problems with the application rather than characterizing and understanding an application's performance behavior. This subtle difference in purpose differentiates this work from standard post-analysis tools. Rather than highlighting hotspots of communication in a particular application for example, this work might focus on the fact that communication in a particular code region is proportionally lower than that of a previous run of the application. In a sense, the focus is on identifying key differences and, in particular, idiosyncratic behaviors.

The approach described in this paper uses a combination of static and dynamic call-graph analysis to identify regions of code that are sensitive to data-set and scalability variations in order to considerably reduce the time-to-model. It provides a compact view of multiple executions of an application using color cues

to draw attention to “areas of interest”. It can also generate graphical illustrations of differences between execution traces to rapidly identify code blocks that require attention. In addition, events can match communication patterns against a library of predefined *templates* that attempt to identify common parallelization strategies. The current implementation is a prototype: it is envisaged that other methods of visualization could be employed to summarize more detailed performance characteristics “at a glance”. Reducing the physical screen estate that represents key data could permit rapid large-scale analysis. Similar visualization techniques have been applied to code maintenance [10].

The paper is organized as follows: Section 2 describes an approach that leads to a performance model and identifies areas where tools can assist. Section 3 introduces a tracing tool that can create call-graphs for post-analysis using source-code instrumentation. In Section 4, methods that use multi-trace visualizations to highlight key application differences are described. Their use is illustrated with examples run on large-scale parallel codes. Section 5 summarizes the features of the approach. Conclusions and future work are discussed in Section 6.

## 2. Identification of performance characteristics from multiple executions

The performance modeling work at Los Alamos National Laboratory (LANL) has been primarily focused on applications representative of the ASCI (Accelerated Strategic Computing Initiative) workload where analytical techniques are employed to develop entire-application models for the large-scale ASCI computing resources. This differs from a number of other performance initiatives that tend to focus on smaller applications in distributed computing environments such as [11]. The Los Alamos models are used most prominently to explore the scaling behavior of applications on existing and speculative future architectures.

The approach to developing a performance model is based upon a detailed understanding of the performance effects that occur when changes to the system and application configuration are applied. Features that effect performance include: the data decomposition, the frequency and locality of boundary communication, the frequency and impact of the collective opera-

tions, and the portion of workload assigned to a particular processor. To obtain an initial behavioral pattern for these features, the preliminary stages of model formulation include executing the application with fixed input data sizes and a varying number of processing elements (PE) to observe changes in the overall execution time and resource consumption. This can reveal basic information, a simple example being whether the program scales weakly or strongly. Likewise, similar observations can be taken by fixing the number of PEs and varying the data-set sizes.

Instrumentation and profiling are subsequently used to obtain a deeper understanding of the code. From a performance-perspective, it is usually unnecessary to understand the “meaning” of the data types used, however the “representation” of the data types can have a large impact on the code’s performance. Identifying the dimensionality of the data and the workload per grid-point also has a large impact on the achievable performance. A number of these performance features can be observed by highlighting the changes between different “executions” of the application in various configurations. Changes may be observed in the communication-to-computation ratio, the communication matrix (which processors are involved in communication), and also in scaling effects. Moreover, observed changes can provide insight into the type and method of domain decomposition such as message size changes as in the case of SAGE [12] (an adaptive mesh hydro code) due to its 1D decomposition, and into differences in neighboring processors as in the case of Tycho [13] (a deterministic radiation transport code) due to the use of an unstructured mesh.

The problem partitioning and related messaging revealed through instrumentation typically provides a strong indication of the arrangement of the application’s data, which can be confirmed by static code analysis of the relevant regions. Generating static call-graphs can assist with identifying the functional dependencies of the code sections and dynamic call-graphs (through the use of traces collected at run-time) can illustrate the flow of execution. In addition, comparing the relative number of issued instructions for a given subroutine in different runs/iterations can highlight the impact of configuration change on calculation and computational areas.

The overall objective of these activities is to obtain a model based on timings of sequential elements

parameterized by expressions that are subject to input parameters and differing levels of parallelism. By identifying messaging, data placement and computational sensitivity to configuration and data-set sizes, it is possible to locate regions of the code that can be described by a single timing (for a given architecture) or by an analytical expression that captures the computation/communication characteristics with respect to the input parameters. While the modeling approach is based on understanding the application's behavior, it is evident that much of the initial work is based upon observing the performance effects of input and configuration variation. This leads to the construction of an initial model, which is subsequently refined until a satisfactory level of fidelity (predicted execution time versus measured time) is obtained.

Even semi-automated approaches that use micro-benchmarking typically require manual tuning to improve predictive accuracy and this inevitably dictates running the application in various configurations and analyzing the results. It is, therefore, reasonable to utilize tools that assist in the early stages of performance model development. A suitable tool can assist with instrumentation and call-graph generation producing a visualization of the key differences to indicate "areas of interest". This can significantly reduce the overhead of the performance modeling process, allowing a performance specialist to focus on a subset of the application rather than all of it.

### 3. Dynamic call-graph collection

In order to assist the instrumentation, an automatic source-to-source translation tool is used to insert profiling statements into the code where a subroutine is entered and exited. This tool currently supports Fortran, parsing the source and inserting profiling statements but could be extended for further language coverage. The application is linked against a lightweight profiling library that stores the "events" in a dynamic, page-based list whenever a subroutine is called. To minimize memory overhead, each event records a limited amount of information which includes: a source-file identifier, a source-line number, and a token to denote whether a "context" has been entered into or exited from. As with nested subroutines, contexts are grouped so that when, for instance, subroutine main

calls subroutine `init` the instrumentation library is in the context of both `main` and `init`, and any events that occur are associated with both of these contexts. This property is used to reduce the storage space of the event list as it is dynamically constructed in memory. The event-list is written to file when the application exits, although it is possible to allow on-line paging. The trace-file can then be processed by subsequent utilities.

During application initialization, the instrumentation library is disabled which results in virtually no overhead. An explicit `PROFILE-START` call is required to enable the library and store the context changes as necessary. In addition to context changes, MPI calls are logged as events using the profiling MPI (PMPI) interface, which utilizes the profile library, and can assist with identifying the communication patterns that occur. The relevant parameters (source, destination, collective, size, type) are stored as part of an MPI event. Currently, only 20 MPI calls are wrapped which covered our internal test cases; it is reasonably straightforward to include further commands or to employ a third party tool such as VampirTrace [9]. A `PROFILE-STOP` can be used to stop the process prematurely (otherwise it occurs on application exit). Each processor logs its own events separately and the logs are written to different files. This might include local scratch disk or a centralized node over a file sharing mechanism if required.

The process allows rapid instrumentation of the source code with a subsequent compile, link and execute sequence to obtain the required tracing information. The events that appear are essentially a call-graph of the program and can be subjected to a wide variety of post analysis tools. This includes the post processing multi-trace analysis tools described here that attempt to identify key performance features.

Inevitably, the application incurs overhead through use of the tracing library although this overhead is kept minimal, consisting of a few array operations per profile call to store events. Depending upon the page size, periodic memory allocation is required which will incur a further performance penalty. It is important to note that the purpose of this task is to identify functional characteristics, and so the timing issues are secondary in the analysis. Furthermore, with judicious use of the profile start and stop functions, it is possible to isolate a single iteration of the code in order to reduce

the size of the trace files, or create a set of trace files for each iteration and processor. This was used when applying these tools to the Los Alamos Parallel Ocean Program (POP) [14] in order to compare different iterations of the application. Where trace file size is not an issue, particular iterations can be extracted during post-processing.

In a similar manner to Paradyn [15], the instrumentation library utilizes the concept of inclusive and exclusive metrics, where inclusive sums a metric for a context and all its children, while exclusive returns the metric for a context without its children. Current recordable metrics include context duration and the number of issued instructions (if PAPI [16] is available). The context duration should be used cautiously as timing anomalies are possible when using a profiling library as well as experimental variations that are inevitable with very short timings. However, the issued instruction count can be more telling about a context's behavior with regard to computational overhead. In isolation, these quantities do not provide much information, unless the purpose of the instrumentation activity is to identify contexts with heavy computational requirements. It is in multi-trace analysis, where one context is compared with another (with a different configuration, or from another processor or iteration), that the relative difference in issued instructions can reveal workload differences. With a fixed workload size, and varying processor count, it is possible to use this technique to identify where the bulk of the sub-grid computation is occurring, as one can reasonably postulate that such contexts will be sensitive to a configuration change of this nature.

#### 4. Multi-trace visualization

There are a number of visualization tools that can assist with understanding an application's runtime behavior. For MPI applications, Vampir provides a large suite of views that provide a good level of detail. It is essentially possible to “playback” the application to determine where communication occurs, which processors are involved and where blocks of computation occur. The tools developed as part of this work provide similar capabilities, although they are geared towards the accompanying tracing library and focus on highlighting features that readily facilitate performance-model generation. These tools are now explored in detail.

##### 4.1. Context-chain graph

The trace fragment depicted in Fig. 1, taken from a run of the Parallel Ocean Code (POP), simply graphs the application flow (in common with a number of existing profilers). However, capabilities exist in the implementation that allows a performance modeler to focus on a particular section of code. In this fragment, the extract is from the main iteration in POP. The first two columns list the instantaneous time and accumulated time within a context, respectively. The third column lists the relative inclusive-percentage of issued instructions, and the remaining columns detail the contexts. The bracketed numbers after each context relate to source line ranges. This example illustrates that a significant amount of computational time is spent in the baroclinic context.

Time	Acc.Time	Inst.	Context
0.20522	00.00001	00.001	timers [152-174]
0.20524	00.00599	99.998	step_mod [47-430]
0.20525	00.00001	00.000	step_mod [47-430] -> timers [96-119]
0.20526	00.00001	00.000	step_mod [47-430] -> timers [96-119]
0.20527	00.00004	00.002	step_mod [47-430] -> forcing [139-184]
0.20528	00.00001	00.000	step_mod [47-430] -> forcing [139-184] -> forcing_ws [383-456]
0.20529	00.00001	00.000	step_mod [47-430] -> forcing [139-184] -> forcing_shf [619-743]
0.20530	00.00001	00.000	step_mod [47-430] -> forcing [139-184] -> forcing_sfwf [536-652]
0.20532	00.00001	00.000	step_mod [47-430] -> forcing [139-184] -> forcing_ap [363-427]
0.20533	00.00001	00.000	step_mod [47-430] -> forcing [139-184] -> forcing_coupled [194-219]
0.20909	00.00002	00.001	step_mod [47-430] -> timers [152-174]
...			
0.20912	00.05369	74.095	step_mod [47-430] -> baroclinic [115-410]
0.20928	00.00002	00.679	step_mod [47-430] -> baroclinic [115-410] -> vertical_mix [249-290]
0.20929	00.00230	00.678	step_mod [47-430] -> baroclinic [115-410] -> vertical_mix [249-290] -> vmix_rich

Fig. 1. Trace fragment for iteration 0, PE #0 of an execution of POP.

F (C)	Context
020	global_reductions [1002-1077] -> .. -> step_mod [47-430] -> MPI_ALLGATHER
140	global_reductions [1002-1077] -> .. -> step_mod [47-430] -> MPI_ALLGATHER
002	global_reductions [510-550] -> .. -> step_mod [47-430] -> MPI_ALLREDUCE
020	global_reductions [1002-1077] -> .. -> step_mod [47-430] -> MPI_ALLGATHER
004	boundary [210-384] -> step_mod [47-430] -> MPI_Irecv from 2 to 0 (88 elements)
004	boundary [210-384] -> step_mod [47-430] -> MPI_Isend from 0 to 1 (88 elements)

Fig. 2. Fragment of the event-count view of the communication in POP.

Each event in this view can be consolidated to provide a compact representation of a program's functionality. It is possible to filter out detail such as in Fig. 2 where only communication events are included in the trace fragment. In this view, the first column denotes the event count with the context-chain on the right extracted from the context profile.

This representation is useful to indicate the mix of activities in the application such as the ratio of collectives and point-to-point communications. As this can have a significant impact on the scalability of a code, it is useful to ascertain the frequency and location of such operations. It can also be used to identify behaviors that are unexpected, such as irregular or unusual communications. For example, the POP application utilizes a solver subroutine that tests for convergence of a particular variable. This leads to an “unexpected” number of MPI-send operations – 97 in this case. Again, identifying unusual behavior can assist the performance specialist in analyzing a particular piece of code in more detail.

The event-count view also lends itself to the analytical expressions that typically constitute a performance model. Capturing the cost of performance characteristics, as opposed to the functional operations, reduces the complexity of the resultant models, as they do not require temporal structure. The event-count view reflects this as it is essentially unimportant when the call is made: just how many times, how often, and any size information. Inevitably, there are applications where the interactions may not follow this pattern but in many situations similar communications (such as boundary exchanges) can invariably be grouped together into a single, simpler expression.

#### 4.2. Context comparison views

Where the approach becomes much more powerful is where comparisons are made between different

application executions. So-called multi-trace analysis highlights the side-effects of running the application with different processors, input data and/or workload size. The tool developed in this work offers a number of facilities to help with such analysis; the first is the ability to place two or more traces “side-by-side”, highlighting the difference between code densities (number of instructions in a subroutine) for differing application-runs (presumably with different input-data sizes). This approach is useful when attempting to identify performance-effected regions of code, as a performance modeler can determine if a code region in an application is worth analyzing further or whether a single timing can be taken.

After parsing an event file, as generated by the instrumentation library described in Section 3, a post-processor reconstructs a complete call-graph of the application and compares it with subsequently loaded event files. The implementation utilizes an algorithm that searches for the largest similar code-region within the main code body in a similar manner to the “diff” command-line utility (although the matching is neither textual nor exact).

Context entries in the trace files are deemed similar if the call-graph nodes are equal and that the instruction counts are within a given sensitivity. Communication entries are deemed similar based on rules that can be configured by the user, in one mode the call-graph nodes, the type (send, receive, gather, etc.) and all communication parameters (source, destination, and size) are all equal. A second mode allows slightly more flexibility: the call-graph and type must be equal, but variations are permitted in the source and receiver identifier. The mode is selectable and depends on whether comparisons are being made between processors, iterations or configurations.

The visualization screenshot in Fig. 3 compares three traces from PE #0, #1 and #2 for a single iteration. Each trace file contains approximately 66,000

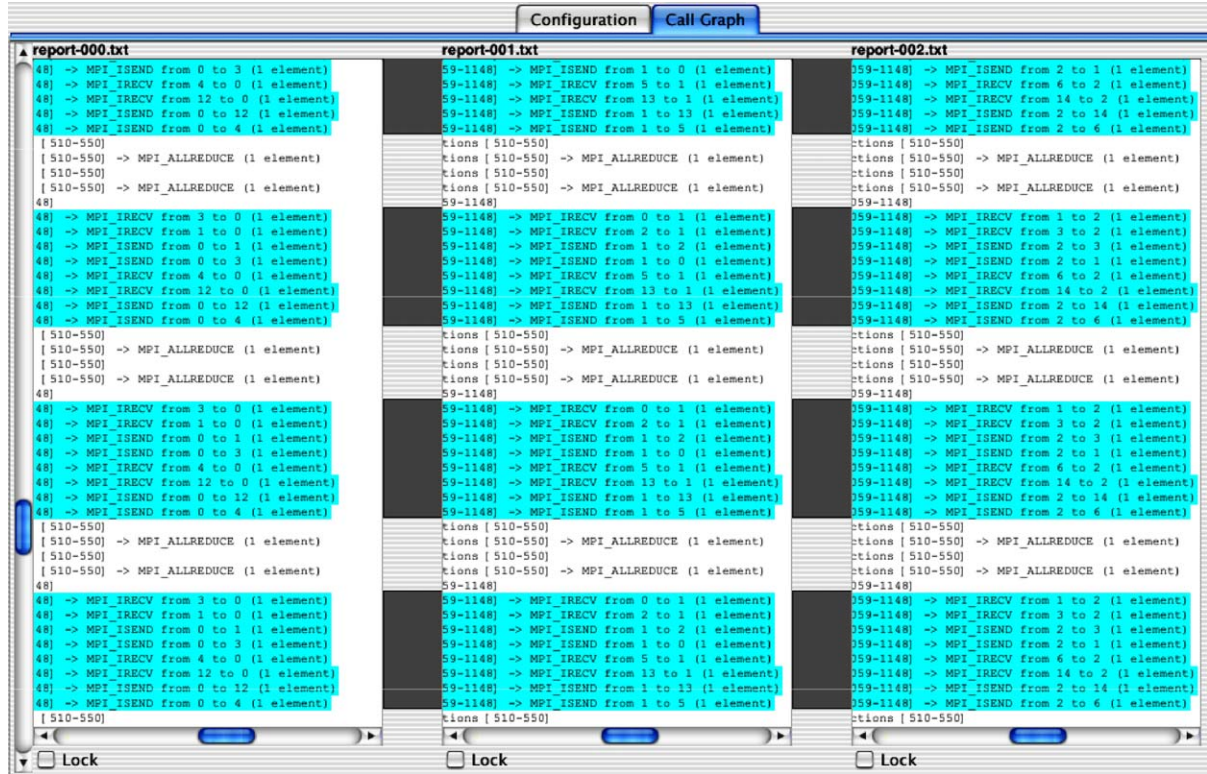


Fig. 3. Multi-trace visualization of three different processors from a single iteration of POP.

events and shaded regions (blue in color) highlight differences. In this particular example, the highlighting indicates the differences between blocks of boundary communication in otherwise similar code regions. These differences are straightforward – that is source and destination pairs are not equal. Using this view (multi-processor), for this application, it is possible to determine that the workload is spread evenly across the sub-grid with regular differences in the communication matrix. While it is not clear from the screenshot, the communication events are arranged within their respective appropriate contexts so that it is straightforward to identify which code blocks are performing the messaging. It also provides indicators on the type of decomposition used in the application, namely that the logical 2D processor mesh is probably cyclic otherwise one might expect that PE #0 would have a fewer communications than PE #1 at the boundaries.

When event-files are loaded, they can be linked together using a control panel to associate relevant view-options. The tool permits some filtering which allows

the user to view MPI collectives, point-to-point functions and normal context (subroutines). A ‘hotspot’ slider allows the user to highlight portions of the trace where the inclusive number of instructions as a percentage of the overall total, is above the threshold percentage. The ‘sensitivity’ slider allows the user to set the extent to which the number of issued instructions in the first trace can deviate from the second. In tests, it was found that a value of approximately 5% highlighted regions of the call graph that altered significantly due to a change in configuration.

The comparison algorithm attempts to identify the largest similar run within the call-graph and creates links to the next trace. Where calls exist in the second trace but not the first they are highlighted using a further color and the link cursor (sited between the trace columns) is used to indicate where they should be in the second trace. Similarly, where calls exist that are not in the second trace, they are highlighted a further color, and again link to where they are expected. A common color (blue) indicates sections that exist in both traces

but are different (i.e. exceed a given threshold). The tool allows a chain of comparisons to be established, so that trace 1 is compared with 2, 2 with 3 and so on. These chains do not have to be compared sequentially – PEs #0 and #2 in Fig. 3 could be compared directly if required.

#### 4.3. Call-chain discrepancy view

Alongside the multi-trace view, it is possible to take one or more trace files and arrange the output so that common elements of the call-chain are “aligned”. This process can produce concise graphic views that summarize major blocks of the application. An example is shown in Fig. 4, a comparison is made between small-scale executions of POP: one on a  $4 \times 4$  (16 processor) arrangement and one on a single processor run. The flow of the code is from the top left to bottom right corner, scanning across each line. Each pixel represents an individual context event, where black denotes that the context-flow is significantly different across traces (such as call-chain variation, or a difference in source/destination PE identifiers), and grey indicates that the chain is identical. The white space between these color cues is a by-product of the visualization algorithm attempting to align black and grey blocks allowing the user to more readily identify related areas of the call graph.

The algorithm itself functions by analyzing the aligned call graphs and constructing sequential chains of matching and non-matching contexts. It scans from top-left to bottom-right, line breaking when the previous chain sequences differs from the current sequence. This results in each horizontal “scan line” containing related activity and, where the application contains loops, grouping this activity vertically.

The advantage of this approach is that many thousands of events are compactly represented using individual pixels (in this figure, each pixel has been scaled up for clarity). When a call-chain discrepancy is detected (black), the filling algorithm checks whether any “similar” discrepancies have occurred on the same line. If so, the line continues from left to right; otherwise the algorithm moves onto the next line, from the left. This groups blocks of common calls together making it easier to pick out particular features.

In Fig. 4, one difference has been highlighted, by the cursor, revealing a difference in the MPI\_Irecv

communication. Moving the mouse over a few of these points rapidly reveals the repetitious nature of this particular difference and clearly demonstrates where the communication is occurring.

Displaying traces at this density allows an observer to rapidly distinguish areas of interest and then “drill down” to view the relevant code. In this case, the striped block relates to a group of boundary exchanges where the processor source/destination pairs are different (the sequential execution clearly has a different boundary exchange pattern than the  $4 \times 4$  case which performs inter-processor exchanges).

As well as applying this technique to different processor configurations, the tool can be used to compare different processors in the same execution run to identify patterns indicative of the data decomposition. It is also possible to consider individual iterations, where call-chain analysis can reveal distinct phases in the application such as the wave-front processes in Sweep3D—a code that performs deterministic  $S_N$  transport on structural meshes [17]. While this process is useful for identifying “binary” differences, aligned events can be further analyzed to discover changes to workload, messaging size and/or computational impact. The instrumentation library retains an issued instruction counter, which can be used to compare differences across execution traces.

#### 4.4. Computational difference view

The graphical view in Fig. 5 applies to the same traces as Fig. 4; however, communication events have been removed and the relative difference between issued instructions is illustrated as a greyscale.

In this figure, black indicates the largest differences in computation with the greyscales ranging up to this. The initial patterns relate to matrices in the solvers being primed with data. With a fixed problem size, this inevitably reduces the workload per-processor as the PE count is scaled up. This is evident in the computational plot. The diagonal pattern at the top of Fig. 5 reveals a regular pattern in the computation that directly relates to the change in workload. As before, it is possible to hone in on a particular event and see the differences in more detail. In Fig. 5, the solvers context has been identified as having computational sensitivity to the start-up configuration.

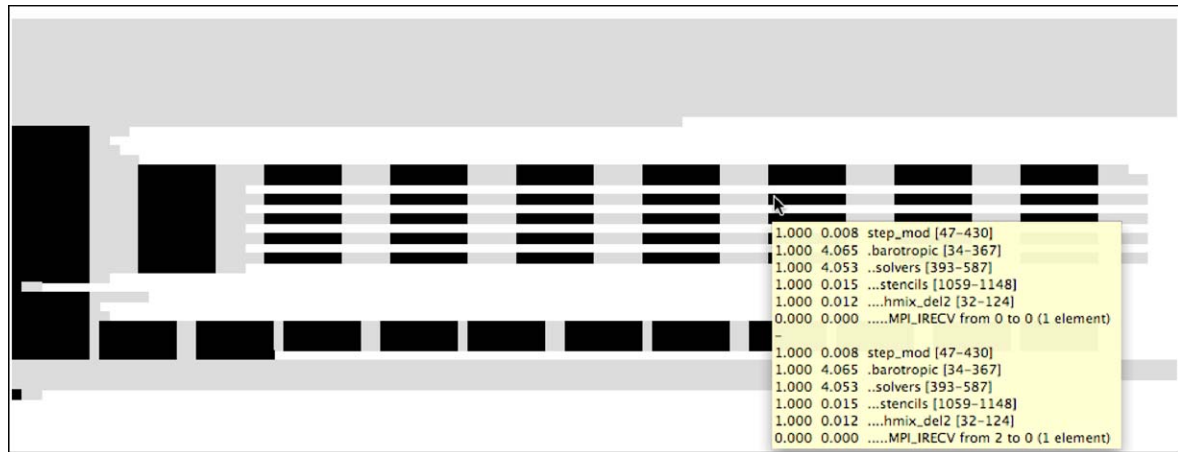


Fig. 4. Communication differences between two POP executions for a  $1 \times 1$  and  $4 \times 4$  processor configuration.

#### 4.5. Communication matrix

Using a communication matrix view, it is possible to identify key features regarding some of the performance features of the POP application, for instance where boundary communications are performed, sensitivity to scaling, computational differences and indicators on the type of domain decomposition employed by the application. Indeed, it is possible to utilize the communication events to hypothesize how the application is representing key data structures. Examination of the communication matrix, a 2D grid of source to destination processor elements can reveal a number of key application behavioral characteristics. Symmetri-

cal patterns are indicative of two-way sub-grid communication, and a nearest neighbor indicates 1D data decomposition for example.

Using 16 traces from a  $4 \times 4$  processor run of the POP application produces the communication matrix illustrated in Fig. 6. The matrix reveals that most PEs communicates with four others and that the communication is two-way due to the symmetrical nature of the plot. This pattern is typical of a 2D decomposition with boundary wrap-round in the X-dimension.

Using the output from the communication call-chains as a basis, the tool developed in this work attempts to identify the decomposition structure using a set of popular communication “templates”. This

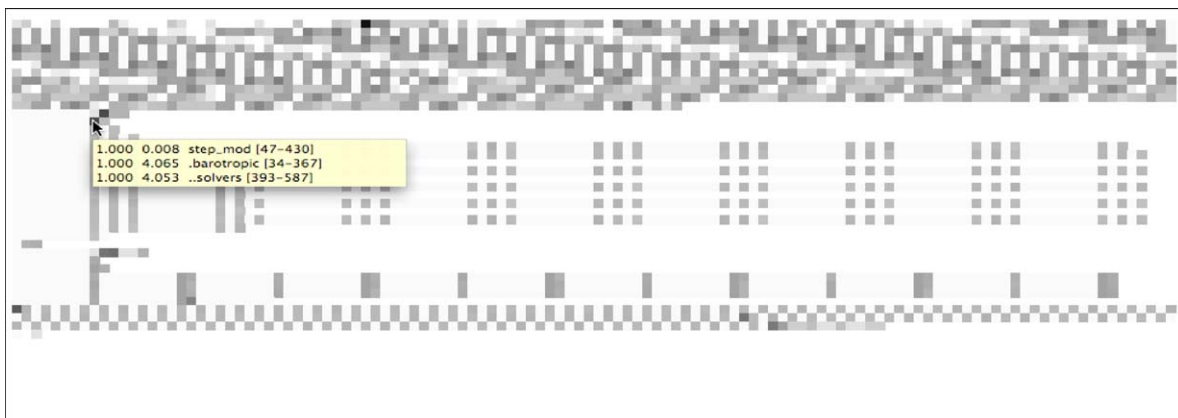
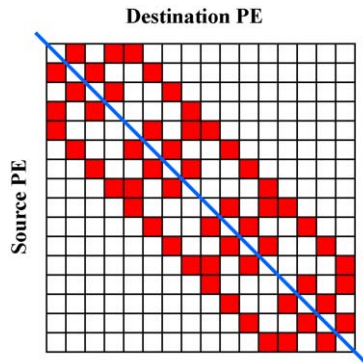


Fig. 5. Differences in issued instructions between a  $1 \times 1$  and  $4 \times 4$  processor executions of POP. The highlighted area is a three columned table that includes a “difference” tag, relative difference between contexts and the context identifier.

Fig. 6. Plot from a  $4 \times 4$  execution of POP.

process can be used to rapidly determine whether an application utilizes “standard” decomposition arrangements, and, when used in combination with the call-graph comparisons, can be used to determine the workload distribution and application data structures.

The templates are defined as a 2D matrix  $T$  representing point-to-point communications between each source processor  $i$  (vertical axis) and each destination processor  $j$  (horizontal axis). Each template element,  $T_{ij}$ , takes a value of  $\{-1, 1\}$  where  $T_{ij} = 1$  if the communication from processor  $i$  to processor  $j$  is a part of the communication pattern represented by the template, and  $T_{ij} = -1$  if it is not. An application communication matrix  $C$  is formed from an analysis of the trace file output, where  $C_{ij}$  is set to 1 if communication occurred between processors  $i$  and  $j$ , or  $-1$  otherwise.

The trace files can be evaluated in their entirety (as in the case of POP) or portions can be analyzed if the application has distinct phases where aggregated messaging can affect the level of detail revealed. Unfortunately, it is application dependent at what level the appropriate amount of detail will be revealed and so obtaining the effective traces that provide useful information relies, to some extent, on the skill and experience of the performance modeler. However, as this process is usually conducted by the modeler, it does not place additional constraints on the approach. Indeed, poor matches may suggest an inadequate number of traces or a trace file that has been truncated and does not contain enough detail to form an exact match.

To start the comparison process, a library of templates are loaded or generated dynamically. The traces are compared with each template in the library using a

Table 1

Communication template matches

$P_x \times P_y$	Bi/Uni	Configuration
$1 \times 16$	BI	NOWRAP = 0.82
$1 \times 16$	BI	WRAP = 0.75
$2 \times 8$	BI	NOWRAP = 0.70
$2 \times 8$	BI	WRAP = 0.69
$4 \times 4$	BI	NOWRAP = 0.94
$4 \times 4$	BI	WRAP = 1.00 (EXACT)
$8 \times 2$	BI	NOWRAP = 0.77
$8 \times 2$	BI	WRAP = 0.75
$16 \times 1$	BI	NOWRAP = 0.82
$16 \times 1$	BI	WRAP = 0.75

convolution (1) that provides a weighted match, where  $p$  is the number of processors. It is possible to run the comparison process with an arbitrary number

$$m = \frac{1}{p^2} \sum_{i=0}^{p-1} \sum_{j=0}^{p-1} T_{ij} \cdot C_{ij} \quad (1)$$

of trace files (or processor counts).

When the communication pattern is regular, this approach is able to discern application behavior, as illustrated in Table 1. This illustrates the matching output when several of the pre-defined templates and using the 16 trace files from a  $4 \times 4$  run of POP. Each of the templates in this example assumed a 2D communication pattern of different logical processor arrangements ( $P_x$  and  $P_y$ ).

In this example, the configuration of POP with a  $4 \times 4$  processor array using bi-directional wrapped boundary exchanges was successfully identified. While the non-wrapped  $4 \times 4$  arrangement was not an exact match, the relatively high score suggests that these communication templates can reveal certain similarities (in this case, the PE configuration) without matching a particular communication strategy.

## 5. Performance feature summary

Together the tools allow a performance modeler to rapidly profile a code, run it under different configurations (such as on a  $4 \times 4$  processor network, and on an  $8 \times 8$  network) and then analyze the traces for particular performance characteristics (see Table 2). In addition, the traces can be analyzed to determine the data decomposition by comparison with a set

Table 2  
Example of the identifiable performance features

Performance characteristic	Tool assistance
Calculate impact due to data-set changes	Highlight relative difference in instruction counts in similar regions of the call-graph Highlight relative difference in message sizes between PEs
Data-placement differences due to variation in PE count	Highlight changes in communication patterns (source, destination), as topology is changed, to account for configuration
Functional differences in application iterations	Highlight new/deleted regions in the call-graph and differences in event-count views

of example communication templates as described in Section 4.

Effective visualizations are used to illustrate key performance differences so that in the case of moving from one processor arrangement to another for example, it is likely that the communication patterns will be different and, if the application scales weakly, large differences in the computation sections and message sizes. These visual cues aim to provide an effective summarized view of the program's dynamic operation. Once alerted to a particular region of code that appears sensitive to a particular change it is possible to direct attention to that part of the program that can ultimately assist in constructing the performance model.

## 6. Conclusions and future work

The approach described in this paper utilizes dynamic trace files and a multi-trace visualization tool to highlight areas of interest when input parameters, data sets and resource configurations are modified. By focusing on the areas of a code that are sensitive to configuration and input data, overhead is removed in terms of isolating the critical regions that govern the performance characteristic of an application.

Analytical modeling is an increasingly important activity in understanding and reasoning about the performance of complex systems [2]. The work presented in this paper aids the performance modeling process by revealing detail through multi-trace analysis that

can *focus* the attention of the performance-specialist to relevant regions of code. The compact “pixel” view permits a rapid overview of the application's behavior that can be annotated in a manner that highlights performance-effected contexts.

Future developments of this approach include developing the visualization tool to utilize the Vampir trace facility that is widely utilized in parallel performance studies. Further work includes experiments with more exotic forms of visualization, using color and locality to obtain very compact views of multiple trace files relating to a number of difference performance scenarios.

## Acknowledgement

Los Alamos National Laboratory is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36.

## References

- [1] D.J. Kerbyson, A. Hoisie, H.J. Wasserman, Verifying large-scale system performance during installation using modeling, in: High Performance Scientific and Engineering Computing Hardware/Software Support, Kluwer, 2003, pp. 43–156.
- [2] F. Petrini, D.J. Kerbyson, S. Pakin, The case of the missing supercomputer performance: achieving optimal performance on the 8192 processors of ASCI Q, in: Proceedings of IEEE/ACM Supercomputing, Phoenix, AZ, 2003.
- [3] D.J. Kerbyson, A. Hoisie, H. Wasserman, A comparison between the earth simulator and alphaserver systems using predictive application performance models, in: Proceeding of IPDPS, Nice, France, 2003.
- [4] D.P. Spooner, S.A. Jarvis, J. Cao, S. Saini, G.R. Nudd, Local grid scheduling techniques using performance prediction, in: IEE Proceeding—Computers and Digital Techniques, vol. 150(2), IEE Press, 2003, pp. 87–96.
- [5] S.C. Perry, R.H. Grimwood, D.J. Kerbyson, E. Papaefstathiou, G.R. Nudd, Performance optimisation of financial option calculations, Elsevier, Parallel Computing 26 (5) (2000) 623–639.
- [6] G. Marin, J. Mellor-Crummey, Cross-architecture performance predictions for scientific applications using parameterized models, in: Proceeding of Sigmetrics, New York, June, 2004, pp. 2–14.
- [7] A. Snaveley, L. Carrington, A. Purkayastha, et al., A framework for application performance modeling and prediction, in: Supercomputing 2002, Baltimore, MD, 2002.
- [8] H.W. Cain, B.P. Miller, B.J.N. Wylie, A callgraph-based search strategy for automated performance diagnosis, in: Concurrency and Computation: Practice and Experience, vol. 14, 2002, pp. 203–217.
- [9] <http://www.pallas.com/e/products/vampir/index.htm>.

- [10] S.C. Eick, J.L. Steffen, E.E. Sumner Jr., Seesoft—a tool for visualizing line oriented software statistics, Special Issue on Software Measurement Principles, Techniques and Environments, *IEEE Trans. Software Eng.* 18 (11) (1992).
- [11] G.R. Nudd, D.J. Kerbyson, et al., PACE: a toolset for the performance prediction of parallel and distributed systems, *Int. J. High Perform. Comput. Appl.* 14 (3) (2000) 228–251.
- [12] D.J. Kerbyson, H.J. Alme, A. Hoisie, F. Petrini, H.J. Wasserman, M.L. Gittings, Predictive performance and scalability modeling of a large-scale application, in: *Proceedings of Supercomputing*, Denver, 2001.
- [13] D.J. Kerbyson, A. Hoisie, S.D. Pautz, Performance modeling of deterministic transport computations, in: *Performance Analysis and Distributed Computing*, Kluwer, 2003, pp. 21–39.
- [14] <http://climate.lanl.gov/>.
- [15] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.E. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall, The paradyn parallel performance measurement tool, *IEEE Comput.* 28 (11) (1995) 37–46.
- [16] S. Browne, J. Dongarra, N. Garner, G. Ho, P. Mucci, A portable programming interface for performance evaluation on modern processors, *Int. J. High Perform. Comput. Appl.* 14 (3) (2000) 189–204.
- [17] A. Hoisie, O. Lubeck, H.J. Wasserman, Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications, *Int. J. High Perform. Comput. Appl.* 14 (4) (2000) 330–346.



**Daniel Spooner** is a lecturer in the Computer Science Department at the University of Warwick. His main research interests are in the areas of high performance systems, analytical performance models and their application in grid and distributed computing. He is currently involved in the UK e-Science programme developing performance-aware middleware services.



**Darren Kerbyson** is currently a researcher in the Performance and Architecture Lab at Los Alamos. Prior to this he was a senior Lecturer in Computer Systems at the University of Warwick in the UK. He has been active in the areas of performance modeling, parallel and distributed processing systems, and image analysis for the last 15 years. He has worked on many performance orientated projects funded by the European Esprit program, UK Government, ONR, and

DARPA. He has published over 70 papers in these areas and has taught courses at undergraduate and postgraduate levels as well as supervising numerous PhD students. He is currently involved in the modeling of large-scale applications on current and future supercomputers at Los Alamos. He is a member of the ACM and the IEEE.